Debugging

With GDB

Bugs

- Universal truth about bugs
  - *Bugs can neither be created nor it can be destroyed, it can only be transformed.*

    - internet

  - *If debugging is the process of removing bugs, then programming is the process of inserting them.*

    - internet

Bugs

- Bugs are usually messy.
- Bug tracking, identification and removal can consume huge amount of programmers time
- Types of bugs
  - Specifications error
  - Design error
  - Programming error
    - Syntax or Compile time error
    - Logical or Semantic error
    - Runtime error or exceptions

General Debugging Techniques

- Typical approach would be to run the program.
- Check the output
- See what went wrong
- Fix the code
- Compile and re-test the program

Five Stages of Debugging

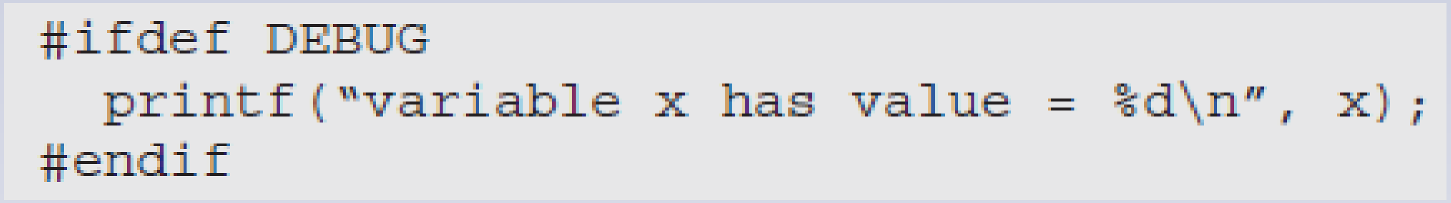
- **Testing:** Finding out what defects or bugs exist
- **Stabilization:** Making the bugs reproducible
- **Localization:** Identifying the line(s) of code responsible
- **Correction:** Fixing the code
- **Verification:** Making sure the fix works

Techniques

- Analyzing the program with bug(s)
  - Cause the program to exhibit a failure
- Code Inspection
  - Or code analysis is a group of developers walking through the entire source code carefully analyzing every line of code
- Instrumentation
  - Adding additional code to the program to collect additional real-time info about the running program.
- Controlled Execution
  - Using debuggers to perform live analysis of the running program

Instrumentation

- Two ways of instrumentation
  - Static Way (of the preprocessor way)
  - Dynamic way
- In the static way, a macro is used to selectively include the instrumentation code.
- Recompilation is required



```
#ifdef DEBUG
  printf("variable x has value = %d\n", x);
#endif
```

- Compile the code with –DDEBUG switch declare the macro to include the instrumentation code or without the macro to exclude it

Instrumentation

- Several macros defined by the preprocessor aid in debugging

Macro	Description
__LINE__	A decimal constant representing the current line number
__FILE__	A string representing the current file name
__DATE__	A string of the form "Mmm dd yyyy", the current date
__TIME__	A string of the form "hh:mm:ss", the current time

Instrumentation

```
// CInfo.c
#include <stdio.h>
int main()
{
#ifdef DEBUG
    printf("Compiled: " __DATE__ " at " __TIME__ "\n");
    printf("This is line %d of file %s\n", __LINE__, __FILE__);
#endif
    printf("hello world\n");
    exit(0);
}
```

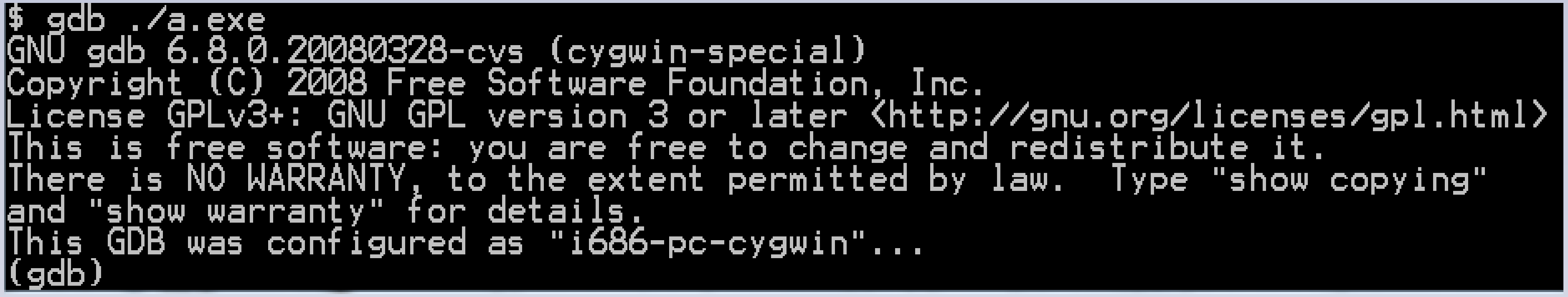
```
$ gcc -o cinfo -DDEBUG cinfo.c
$ ./cinfo
Compiled: Jan 9 2012 at 09:15:22
This is line 7 of file cinfo.c
hello world
$
```

Instrumentation

- Dynamic Way of Instrumentation
- Instead of a macro, a global variable can be created to act as a flag
- Based on the flag content, the instrumentation code can be either executed or skipped
- And without rebuilding the code.
- A provision should be provided to set that flag
  - Either through command line switch
  - Or through any gui dialog boxes
- The instrumentation code should always dump the output to the stderr or to a external logging facility.

Debugging with gdb

- gdb is a very capable debugger available on many unix and linux distros
- Gdb is started by gdb command



```
$ gdb ./a.exe
GNU gdb 6.8.0.20080328-cvs (cygwin-special)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-pc-cygwin"...
(gdb)
```

- Type help at the gdb prompt to get the list of all commands
- gdb itself is a text based program and provides some useful shortcuts for repetitive work - stepping into

GDB

- To work with gdb effectively, the program being debugged should contain debug symbols.
- This is achieved by –g switch during compilation
  - **g++ -g -o myapp myapp.cpp**
- Symbols provide the necessary information about the process being debugged

Starting GDB

- Start GDB using ***gdb*** command at the prompt
- Type quit (q) or Ctrl+d to quit the gdb
- The more usual way of starting the gdb is specifying the program to debug
  - gdb *program*
- Another way of starting the gdb is to specify the program and the core file
  - gdb *program core*
- And gdb can also be used to debug a running process by specifying the program and the process id
  - gdb program 1234
- Arguments to the debugee can also be passed
  - gdb **–args** program a1 a2 a3

Core dump

- System generates a core dump file in the same folder as of the executable if the application crashes
- Core dump wont be generated if the user limit for core is not set.
- By default, the user limit for core dump is 0.
- Set the limit by using the ***ulimit*** command
  - $ ulimit –c 2000
- -c switch sets the core dump limit in KiB

GDB Commands

- GDB commands can be abbreviated to first few characters of the command
- Those first few characters should not be ambiguous
- The <TAB> key can be used to complete the partially typed command name
- GDB command is a single line of input along with optional arguments for the command itself.
- The command itself can be of any length

Gdb: help command

```
(gdb) **help**
List of classes of commands:
aliases — Aliases of other commands
breakpoints — Making program stop at certain points
data — Examining data
files — Specifying and examining files
internals — Maintenance commands
obscure — Obscure features
running — Running the program
<and many more lines>
```

Running GDB

- The **run** command is used to start the debugging of the program
- The **attach** command attaches to the program already running (The program started outside gdb)
- The **detach** command is used to detach from the attached process.
  - The detached process continues to run
  - GDB and the debugee becomes independent
- If the **run** command is executed or quit gdb while still attached to the program, the program is killed

Stopping and Continuing

- The principle purpose of the debugger is to stop the program before it terminates and and investigate the problem
- Inside gdb, the program may stop for any of several reasons, such as
  - A signal,
  - A breakpoint
  - Or reaching a new line after a gdb command such as step.
- You may then examine and change variables, set new breakpoints or remove old ones, and then continue execution.
- ***info program*** command displays the info about the program being debugged

Breakpoints, Watchpoints and Catchpoints

- A breakpoint makes your program stop whenever a certain point in the program is reached.
- For each breakpoint, conditions may be added to control them in finer detail
- A watchpoint is a special breakpoint that stops your program when the value of an expression changes.
- A catchpoint is another special breakpoint that stops your program when a certain kind of event occurs.
  - Such as the throwing of a C++ exception or the loading of a library

Breakpoints, Watchpoints and Catchpoints

- gdb assigns a number to each breakpoint, watchpoint, or catchpoint when it is created.
- These numbers are successive integers starting with one.
- In many of the commands for controlling various features of breakpoints, the breakpoint number is used to specify which breakpoint needs to be changed
- Each breakpoint may be enabled or disabled;
  - If disabled, it has no effect on your program until you enable it again.

Setting break points

- Breakpoints are set with the break command (abbreviated b).
- ***break function*** places a break point on a function
  - Programs are to be built with –g option
- ***break +offset***
- ***break –offset*** places the break point at some address above or below the current address of execution
- ***break line-no*** places the break point on a line number of the current source
- ***break filename:line-no*** places the break point on a line no. in a specific file

Setting break points

- ***break filename:function*** places a breakpoint at a particular function in a specified file.
- ***break *address*** places a break point at a specified address. This command is used when the program doesn't have debug info
- ***break*** without any arguments will place the break point on the next instruction

Setting watch points

- A watchpoint may be set to stop execution whenever the value of an expression changes without having to predict a particular place where this may happen.
- Depending on your system, watchpoints may be implemented in software or hardware.

- ***watch expr*** Set a watchpoint for an expression. gdb will break when expr is written into by the program and its value changes.
- ***rwatch expr*** Set a watchpoint that will break when watch expr is read by the program.
- ***awatch expr*** Set a watchpoint that will break when expr is either read or written into by the program

Setting watch points

- gdb sets a hardware watchpoint if possible.
- Hardware watchpoints execute very quickly,and the debugger reports a change in value at the exact instruction where the change occurs.
- If gdb cannot set a hardware watchpoint, it sets a software watchpoint, which executes more slowly and reports the change in value at the next statement, not the instruction, after the change occurs.

Setting catch points

- Catchpoints can be used to cause the debugger to stop for certain kinds of program events, such as C++ exceptions or the loading of a shared library.
- **catch *event*** is used to setup the catch point
- The event can be
  - **throw** - The throwing of a c++ exception
  - **catch** - The catching of the c++ exception
  - **exec –** The catching of the calls made to exec()
  - **fork –** The catching of the calls made to fork()

Deleting a break point

- It is often necessary to eliminate a breakpoint, watchpoint, or catchpoint once it has done its
- This is called deleting the breakpoint.
- With the clear command you can delete breakpoints according to where they are in your program.
- With the delete command you can delete individual breakpoints, watchpoints, or catchpoints by specifying their breakpoint numbers.

Deleting a break point

- ***clear*** - Delete any breakpoints at the next instruction to be executed in the selected stack frame
- ***clear function***
- ***clear filename:function*** **-** Delete any breakpoints set at entry to the function function.
- ***clear linenum***
- ***clear filename:linenum*** **-** Delete any breakpoints set at or within the code of the specified line.
- ***delete [breakpoints] [range...]*** - Delete the breakpoints, watchpoints, or catchpoints of the breakpoint ranges specified as arguments.
  - If no argument is specified, delete all breakpoints

Enabling and Disabling a breakpoint

- Instead of deleting the break point (or other points), they can be disabled.
- Disabled breakpoints are inoperative until they are enabled back
- ***disable [breakpoints] [range...]*** **-** Disable the specified breakpoints or all breakpoints
- ***enable [breakpoints] [range...]*** - Enable the specified breakpoints (or all defined breakpoints).

Conditional Break points

- A breakpoint with a condition evaluates the expression each time your program reaches it, and your program stops only if the condition is true.
- Break conditions can be specified when a breakpoint is set, by using '***if***' in the arguments to the break command.
- They can also be changed at any time with the ***condition*** command.
- (gdb) break foo if x>0

Breakpoint command list

- A break point can be configured to execute a series of commands when it breaks.
- The list of commands are placed with the commands and End block

```
break foo if x>0
commands
    silent
    printf "x is %d\n",x
    cont
end
```

Continuing and Stepping

- Continuing means resuming program execution until your program completes normally.
- In contrast, stepping means executing just one more "step" of the program,
  - Where "step" may mean either one line of source code, or one machine instruction
- ***continue*** – resumes the execution of the program
- ***step*** - Continue running your program until control reaches a different source line, then stop it and return control to gdb.

Continuing and Stepping

- ***next*** **-** Continue to the next source line in the current (innermost) stack frame. This is similar to step, but function calls that appear within the line of code are executed without stopping.
  - Execution stops when control reaches a different line of code at the original stack level that was executing when you gave the next command
- ***finish*** **-** Continue running until just after function in the selected stack frame returns.

Examining the Stack

- When a program stops, the first thing that is needed to know is where the program has stopped.
- Each time a function is called, the call information is generated.
  - This information is called as "Stack Frame"
- The stack frames are allocated in a region of memory called the call stack.
- When your program stops, the gdb commands for examining the stack allow you to see all of this information.

Stack Frame

- The call stack is divided up into contiguous pieces called stack frames, or frames for short.
- Each frame is the data associated with one call to one function.
- The frame contains
  - The arguments given to the function,
  - The function's local variables, and
  - The address at which the function is executing.

Stack Frame

- The program always starts with a initial frame
  - The frame of the main function (also called as the outermost frame)
- Each time a function is called, a new frame is created and when the function returns, that frame is deleted.
- If a function is recursive, there can be many frames for the same function
- The frame for the function in which execution is actually occurring is called the innermost frame.
- This is the most recently created of all the stack frames that still exist.
- The stack frames are identified by stack frame pointer register

Stack Frame

- ***frame args*** **-** The frame command allows you to move from one stack frame to another, and to print the stack frame you select.
  - args may be either the address of the frame or the stack frame number.
  - Without an argument, frame prints the current stack frame.
- ***select-frame*** **-** The select-frame command allows you to move from one stack frame to another without printing the frame.
  - This is the silent version of frame.

Stack backtrace

- A backtrace is a summary of how your program got where it is.
- It shows one line per frame, for many frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack.
- ***backtrace***
- ***bt*** - Print a backtrace of the entire stack: one line per frame for all frames in the stack.

Stack backtrace

- ***up n*** - Move n frames up the stack.
- ***down n* -** Move n frames down the stack.
- ***Info frame*** – Prints the verbose information about the current frame
- ***info args* -** Print the arguments of the selected frame, each on a separate line
- ***info locals*** **-** Print the local variables of the selected frame, each on a separate line.

Examining the source

- gdb can print parts of your program's source, since the debugging information recorded in the program tells gdb what source files were used to build it.
- When the program stops, gdb will display the source line where it stopped
- ***list linenum*** - Print lines centered around line number linenum in the current source file.
- ***list function*** **-** Print lines centered around the beginning of function function.
- ***List*** **-** Print more lines.

Source and Machine Code

- The ***info line linespec*** commad can be used to map the source line to program addresses and vice versa.
- The ***disassemble*** command is used to display large assembly instructions of the program
- The ***info registers*** command displays the content of all the registers along with the register name.
  - The $[regname] as convenience variables
  - $sp provides the content of stack pointer register

Examining Data

- The usual way to examine data in your program is with the ***print*** command (abbreviated p)
- It evaluates and prints the value of an expression of the language the program is written in.
- ***print expr***
- ***print /f expr*** - expr is an expression (in the source language).
  - By default the value of expr is printed in a format appropriate to its data type; a different format can be chosen by specifying '/f', where f is a letter specifying the format

Examining Data

- The format letters supported are:
  - ***x*** - Regard the bits of the value as an integer, and print the integer in hexadecimal.
  - ***d*** - Print as integer in signed decimal.
  - ***u*** - Print as integer in unsigned decimal.
  - ***o*** - Print as integer in octal.
  - ***t*** - Print as integer in binary. The letter ‘t’ stands for “two”
  - ***a*** - Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol.
  - ***c*** - Regard as an integer and print it as a character constant.
  - ***f*** - Regard the bits of the value as a floating point number and print using typical floating point syntax.

Program Variables

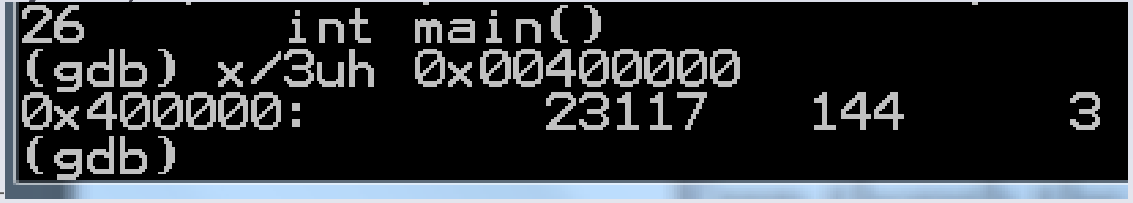
- The most common kind of expression to use is the name of a variable in the program.
- Variables in expressions are understood in the selected stack frame
- They must be either:
  - Global (or file-static)
  - Visible according to the scope rules of the programming language from the point of execution in that frame

Examining Memory

- The command ***x*** can be used for examining the memory in several formats.
- ***x /nfu addr***
- ***x addr***
- ***x* -** n, f, and u are all optional parameters that specify how much memory to display and how to format it;
  - addr is an expression giving the address where you want to start displaying memory

Examining Memory

- ***n***, the repeat count - The repeat count is a decimal integer; the default is 1.
    - It specifies how much memory (counting by units u) to display.
- ***f***, the display format - The display format is one of the formats used by print command
- ***u***, the unit size The unit size is any of
    - b Bytes.
    - h Halfwords (two bytes).
    - w Words (four bytes). This is the initial default.
    - g Giant words (eight bytes).



```
26          int  main()
(gdb) x/3uh 0x00400000
0x400000:         23117    144      3
(gdb)
```

Automatic Display

- If a variable's value should be displayed every time the program stops, then that variable should be added to the Automatic display list
- ***display expr*** **-** Add the expression expr to the list of expressions to display each time your program stops.

This is only the beginning…